VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Komponenta výukového serveru TI - RV a (N)KA za specifikace jazyka

Component of Teaching Server for Theoretical Computer Science - RE and (N)FA from Language Specification

2020 Josef Dostál

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Josef Dostál**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Komponenta výukového serveru TI - RV a (N)KA ze specifikace jazyka
Component of Teaching Server for Theoretical Computer Science - RE and (N)FA from Language Specification

Jazyk vypracování: čeština

Zásady pro vypracování:

V rámci diplomových a bakalařských prací vzniká výukový server pro předměty teoretické informatiky. Jedná se o sadu dynamických webových stránek umožňujících studentům pochopení různých typů úloh a problémů tím, že si mohou zadat na stránce libovolné zadání a zobrazí se jim řešení včetně postupu.

Cílem této bakalářské práce je vytvořit komponentu pro výuku tvorby konečných automatů a regulárních výrazů ze specifikace jazyka vyjádřené matematickým zápisem množiny slov.
1. Vytvořte dynamické webové stránky umožňující uživateli následující:
a) Zadat specifikaci jazyka výběrem z přednastavené množiny typických regulárních vlastností (podslovo, prefix, délka slova, počet výskytů konkrétního znaku, ...) a jejím doplněním o konkrétní hodnoty (abeceda, konkrétní podslovo, minimální nebo maximální hodnota u délky slova apod.).
b) K takto specifikovanému jazyku bude možno si na stránce zobrazit odpovídající konečný automat.
c) Dále bude možné si k jazyku zobrazit odpovídající regulární výraz.
2. Vytvořte i ukázkové vstupy (tedy kompletní specifikace jazyků) tak, aby si vše uživatel mohl vyzkoušet i bez zadávání vlastních vstupů.
3. Použité technologie budou voleny tak, aby byly stránky na straně klienta co nejméně platformně závislé.

Seznam doporučené odborné literatury:

[1] Dexter C. Kozen: Automata and Computability, Springer, 1997
[2] P. Jančar: Teoretická informatika, VŠB-TU Ostrava 2007 (2010)

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Kot, Ph.D.**

Datum zadání: 01.09.2019

Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 14. května 2020 .....

Rád bych tímto poděkovat svému vedoucímu této bakalářské práce Ing. Martinu Kotovi, Ph.D. za odbornou pomoc. Stejně tak bych chtěl vyjádřit díky svým rodičům a manželce Martině za jejich trpělivost a podporu nejen při psaní této práce, ale také v průběhu celého studia.

Abstrakt

Tato bakalářská práce je zaměřena na problematiku tvorby konečného automatu a regulárního výrazu z definice formálního jazyka. Tyto pojmy jsou zde stručně vysvětleny, stejně jako algoritmy či technologie, které jsou k tomu potřeba. Za účelem demonstrace vznikla komponenta výukového serveru teoretické informatiky ve formě webové aplikace, která je schopna podle zadaného nebo náhodně sestaveného jazyka vygenerovat příslušný regulární výraz a nedeterministický konečný automat.

Klíčová slova: Bakalářská práce; konečný automat; regulární výraz; formální jazyk; webová aplikace

Abstract

This bachelor thesis is focused on problematics of creation of a finite automaton and a regular expression from a definition of a formal language. These terms are briefly explained here as well as the algorithms and technologies needed for that. A web application was created for demonstration purposes as a component of an teaching server of theoretical informatics. It is able to generate a nondeterministic finite automaton and regular expression according to a specified or randomly created formal language.

Keywords: Bachelor thesis; finite automaton, regular expression; formal language; web application

Obsah

Seznam použitých zkratek a symbolů

DKA	–	Deterministický Konečný Automat
NKA	–	Nedeterministický Konečný Automat
ZNKA	–	Zobecněný Deterministický Konečný Automat
HTML	–	Hyper Text Markup Language
CSHTML	–	ASP.NET Rezor Web Page
XML	–	Extensible Markup Language
XAML	–	Extensible Application Markup Language
PDF	–	Portable Document Format

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

V rámci výuky teoretické informatiky vzniká výukový server, který má za cíl pomoct studentům při jeho studiu. V této bakalářské práci bude vytvořena a popsána jeho komponenta, zabývající se převodem definice formálního jazyka na regulární výraz a konečný automat. Může sloužit například k procvičení, ale i jako referenční řešení příkladů na toto téma.

Tato aplikace je vyvíjena, mimojiné, i protože nebyla nalezena žádná jiná veřejně dostupná webová aplikace, která umí vygenerovat jak konečný automat, tak regulární výraz z definice formálního jazyka.

Z podobných aplikací však stojí za zmínku například FSM2Regex [1], která je schopna převést regulární výraz na konečný automat (deterministický i nedeterministický) a obráceně.

Další užitečná aplikace byla vytvořena Michaelem Janošíkem v rámci diplomové práce[1], ve které jsou implementovány různé operace s konečnými automaty, jako převod NKA na DKA, průnik či sjednocení dvou automatů a další.

Obě aplikace se však týkají jiné problematiky. Ani jedna nevyužívá pro specifikaci konečných automatů nebo regulárních výrazů definice formálního jazyka. Toto je také důvod, proč je tato práce unikátní a proč má smysl se tímto problémem zabývat.

V sekcích 2, 3 a 4 budou postupně vysvětleny pojmy „formální jazyk“, „konečný automat“ a „regulární výraz“. Sekce 5 rozebírá technologie, které byly ve vytvořené aplikaci použity. Dále, v sekci 6, bude popsána její architektura včetně třídních diagramů knihoven, které v rámci této práce vznikly. V sekci 7 jsou vysvětleny hlavní použité algoritmy. V části „Nasazení“ (sekce 8) je popsáno prostředí, potřebné pro správný chod aplikace a závěrečná sekce (sekce 9) shrnuje práci a možnosti jejího dalšího rozšíření.

[1] Aplikace je dostupná na teoretickainformatika.cz/ka [2]

2 Formální jazyky

Definice formálního jazyka se skládá ze dvou částí. První je abeceda. Značíme ji jako „Σ“ (velká sigma) a určuje, které znaky mohou slova jazyka používat. Mohou to být například znaky latinky, číslice, znaky Braillova písma a podobně[2]. Posloupnosti těchto symbolů nazýváme „slova nad abecedu Σ“ a množinu všech takových slov značíme Σ^*. Patří sem i slovo nulové délky, tedy „prázdné slovo“, které značíme ε (malé epsilon). Můžeme také použít Σ^+, čímž označujeme množinu neprázdných slov nad abecedou Σ. Jazyk, využívající abecedu Σ nazýváme „jazyk nad abecedou Σ“.

Další částí jsou pravidla, která musí slova splňovat, aby do jazyka patřila. Příkladem může být pravidlo, které říká, že každé slovo formálního jazyka musí začínat na písmeno „a“. Slova, která tuto podmínku nesplňují do jazyka nepatří. Nad pravidly můžeme provádět i různé operace, jako sjednocení nebo průnik pravidel, případně negaci pravidla.

Příklad 1
Mějme abecedu $A = \{a, b, c\}$ a jazyk $L = \{w \in A | w$ začíná symbolem $a\}$. Čteme: jazyk L nad abecedou A, kde každé slovo, patřící do tohoto jazyka, začíná symbolem a.

Patří sem například slova *abc*, *a*, *aab* atd. Naopak slova *bbc*, *cba* atd. do jazyka L nepatří. ■

[3]

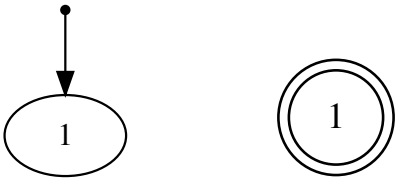
[2] V této práci mohou být však použity pouze alfa-numerické znaky, tzn. písmena - malá i velká - a jednociferná čísla.

3 Konečné automaty

Konečný automat je model primitivního počítače, složeného z konečné množiny stavů, mezi kterými se přechází na základě symbolů, čtených ze vstupu. Jazyku, který je možno zapsat ve formě konečného automatu se říká regulární jazyk.

Konečné automaty definují 2 speciální druhy stavů a to počáteční a přijímající. Počáteční stavy jsou označeny šipkou, která do nich směřuje „odnikud“. Přijímací stavy jsou reprezentovány dvojitým kolečkem.

Konečné automaty lze zapsat několika způsoby. Nejčastějším je však stavový diagram, který je v této práci použit. Jedná se množinu uzlů, reprezentující stavy automatu, a hran, po kterých lze mezi těmito stavy přecházet.



Obrázek 1: Příklad počátečního stavu (vlevo) a přijímacího (vpravo)

3.1 Deterministický konečný automat

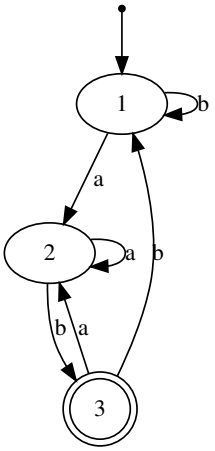
Konečný automat nazýváme deterministický (zkratkou DKA), jestliže je v každé chvíli jeho výpočtu právě jedna možnost pokračování. Lze to zajistit tím, že má právě jeden počáteční stav a ze stavu právě jeden přechod přes každý symbol v abecedě (neboli z každého uzlu grafu automatu vystupuje 1 hrana pro každý znak abecedy).

Příklad takového automatu definující jazyk obsahující slova končících sufixem „ab“ nad abecedou $\Sigma = \{a, b\}$ je znázorněn na obrázku 2.

Definice 1 *Deterministický konečný automat, zkráceně DKA, je dán uspořádanou pěticí* $A = (Q, \Sigma, \delta, q_0, F)$, *kde:*

- Q *je konečná množina stavů*
- Σ *je konečná abeceda*
- $\delta : Q \times \Sigma \to Q$ *je přechodová funkce*
- $q_0 \in Q$ *je počáteční stav*
- $F \subseteq Q$ *je množina přijímacích stavů*

[4]



Obrázek 2: DKA pro jazyk všech slov se sufixem „ab“

3.2 Nedeterministický konečný automat

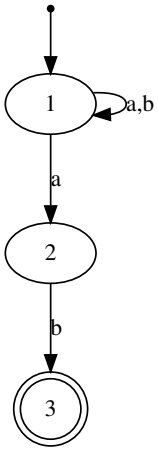
Nedeterministický konečný automat již může obsahovat celou množinu počátečních stavů. Z uzlu může pro každý znak abecedy vycházet libovolné množství hran (ale také z něj nemusí vycházet vůbec). Pokud však neexistuje přechod, po kterém musíme přejít, slovo je automaticky nepřijato.

Jedním z rozšíření je zobecněný nedeterministický konečný automat (zkratkou ZNKA). Mohou se v něm vyskytovat i tzv. ε (epsilon) přechody. Po těch je možno přecházet bez využití znaku, který je právě na vstupu. Jiným rozšířením je automat, jehož hrany nepřijímají jen samotné znaky abecedy, ale celé regulární výrazy. Ten je využit například při převodu konečného automatu na regulární výraz v sekci 7.3.4.

Definice 2 *Nedeterministický konečný automat, zkráceně NKA, je dán uspořádanou pětici* $A = (Q, \Sigma, \delta, I, F)$*, kde:*

- Q *je konečná neprázdná množina stavů*
- Σ *je konečná neprázdná abeceda*
- $\delta : Q \times \Sigma \to P(Q)$ *je přechodová funkce*
- $I \subseteq Q$ *je množina počátečních (iniciálních) stavů*
- $F \subseteq Q$ *je množina přijímajících (koncových) stavů*

[5]



Obrázek 3: Příklad NKA, popisující stejný jazyk, jaký je popsán na obrázku 2

4 Regulární výrazy

Existuje mnoho druhů regulárních výrazů. Vždy se jedná o textový řetězec. V čem se liší je například jejich účel nebo operace, které můžou používat. Většina programovacích jazyků definuje svou vlastní formu. Ty však většinou slouží pro účely jako nacházení nějaké části textu, která lze regulárním výrazem popsat (třeba e-mailová adresa).

V této práci budou využity pro popis jazyků. Použijeme k tomu regulární výrazy s jednou z minimálních množin operací. Těmi jsou:

- **Sjednocení**: Budeme jej zapisovat jako „*výrazA + výrazB*“ a vyjadřuje, že může být vybrán jeden z uvedených výrazů.
- **Zřetězení**: Zapisuje se jako „*výrazA* · *výrazB*“ a říká, že za výrazem A následuje bezprostředně výraz B. Tečka („·“) může být z praktických důvodů vynechána.
- **Iterace jazyka**: Značíme jako „*výraz**“. Zajišťuje, že se výraz může opakovat libovolněkrát.

Dovoleno je také používání závorek, které, stejně jako u aritmetických operací, mění prioritu operátorů. Všechny priority jsou vypsány v tabulce 1.

	$\cdot$	$+$	*	uzávorkovaný podvýraz
$\cdot$		$\gtrdot$	$\lessdot$	$\lessdot$
$+$	$\lessdot$		$\lessdot$	$\lessdot$
*	$\gtrdot$	$\gtrdot$		$\lessdot$
uzávorkovaný podvýraz	$\gtrdot$	$\gtrdot$	$\gtrdot$	

Tabulka 1: Tabulka priorit operací pro regulární výrazy

Jazyk, který lze popsat regulárním výrazem je regulární (lze ho popsat i konečným automatem).

Definice 3 *Induktivní definice regulárních výrazů nad abecedou* Σ*:*

- $\emptyset$*,* ε*,* a *(kde* $a \in \Sigma$*) jsou regulární výrazy:*

  $\emptyset$ ... *označuje prázdný jazyk*
  ε ... *označuje jazyk* $\{\varepsilon\}$
  a ... *označuje jazyk* $\{a\}$

- *Jestliže* α*,* β *jsou regulární výrazy, pak i* $(\alpha + \beta)$*,* $(\alpha \cdot \beta)$*,* (α^*) *jsou regulární výrazy:*

  $(\alpha + \beta)$ ... *označuje sjednocení jazyků označených* α *a* β
  $(\alpha \cdot \beta)$ ... *označuje zřetězení jazyků označených* α *a* β
  (α^*) ... *označuje iteraci jazyka označeného* α

- *Neexistují žádné další regulární výrazy než ty definované podle předchozích dvou bodů*

[6]

Pro přehlednost zápisu se často používají konvence pro vypouštění závorek. Je možné vypustit krajní pár závorek a dále závorky na základě výše uvedených priorit operátorů a asociativity zřetězení a sjednocení. V této práci dále budeme za regulární výraz považovat i řetězec, který vznikne z regulárního výrazu podle definice 3 použitím konvence pro vypouštění závorek.

5 Použité technologie

5.1 .NET core

Tento projekt je postaven na .NET core verze 3.1, což je multiplatformní software-ový framework od společnosti Microsoft. Aplikace v něm napsané mohou běžet jak na operačním systému Windows, tak Linux nebo macOS. Dovoluje také použití různých programovacích jazyků. Tento projekt je však psán v jazyce C#. Framework je open source. Více o tomto nástroji v jeho oficiální dokumentaci [7].

5.1.1 ASP.NET core

.NET core framework nabízí mnoho druhů aplikací. Jedním z nich je ASP.NET core, který se používá pro tvorbu dynamických webových stránek. Více informací je k nalezení na stránkách oficiální dokumentace [8].

5.1.2 Návrhový vzor MVC

Návrhový vzor MVC (Model-View-Controller) se používá především pro tvorbu uživatelských rozhraní.

Jednou z částí jsou pohledy (*Views*), které udávají šablonu, jak bude uživatelské prostředí vypadat. Většinou se pro jejich popis používají různé formy XML souborů (například HTMLCS pro ASP.NET nebo XAML pro WPF).

Další částí jsou modely (*Models*). Jedná se o instance tříd, ukládající data, která jsou zpracovávána. Doplňují se do nich data získaná například z formulářů nebo naopak z jiných částí programu či z databáze.

Posledním pilířem MVC jsou kontrolery (*Controllers*). Ty obstarávají samotnou logiku. Provádí výpočty a dostávají data z uživatelských vstupů, která propagují buď do příslušeného modelu nebo do jiných částí projektu.

5.2 Graphviz

Graphviz je soubor programů, používaných k vykreslování grafů. Je schopen vytvářet obrázky v různých formátech a pomocí javascriptu může být použit i ve webovém rozhraní. Toho bylo využito i zde.

Pro specifikaci vykreslovaného grafu se používá jazyk DOT. Textová reprezentace grafu v tomto jazyce je v programu generována automaticky pomocí metody `GetDot` ve třídě `Graph` (viz podsekci 6.1.1).

5.2.1 Jazyk DOT

V této podsekci bude popsána pouze minimální funkcionalita programu GraphViz a jazyka DOT, potřebná pro tuto práci. Více o programu lze nalézt v jeho oficiální dokumentaci [9].

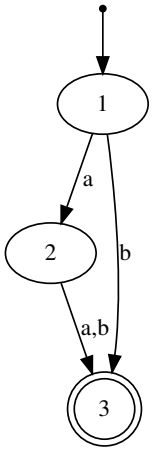
Každý graf musí začínat slovem `digraph`, po kterém následuje název. Tělo grafu se píše do složených závorek. To je složeno ze seznamu uzlů a hran. Každý uzel své označení (to je do něj ve vykresleném grafu vepsáno), za který se do hranatých závorek může upřesnit jeho tvar. V projektu se používá pouze tvar `doublecircle`, pro označení přijímacích stavů a `point`, pro zamaskování uzlu, z něhož vystupuje hrana do nějakého z počátečních stavů (program nenabízí možnost, jak vytvořit hranu „odnikud“ nebo jiný přístup, jak tímto způsobem označit počáteční stavy). Hodnota takového maskovaného uzlu je v projektu náhodně vygenerované GUID.

Nemusí zde však být uvedeny všechny uzly. Pouze ty, u kterých chceme upravit jejich tvar, případně do nich ani z nich nevede žádná hrana. Pokud jsou uvedeny pouze na nějaké straně hrany, jsou vykresleny jako klasický uzel.

V další části se vypisují hrany. U Každé hrany specifikujeme výchozí uzel, tvar hrany, cílový uzel a do hranatých závorek text, který u ní bude. V projektu jsou vykreslovány pouze orientované grafy, takže jsou zde použity pouze hrany zakončené šipkou - „`->`“.

```
digraph G {
   "eff006a3-04b6-4e01-aaeb-ed7e4c0ee1ce" [shape=point]
   1
   2
   3 [shape=doublecircle]

   "eff006a3-04b6-4e01-aaeb-ed7e4c0ee1ce" -> 1
   1 -> 2 [label=a]
   2 -> 3 [label="a,b"]
   1 -> 3 [label=b]
}
```

Výpis 1: Příklad vygenerovaného řetězce pro vytvoření grafu v jazyce DOT



Obrázek 4: Příklad vygenerovaného grafu podle příkladu jazyka DOT ve výpisu 1

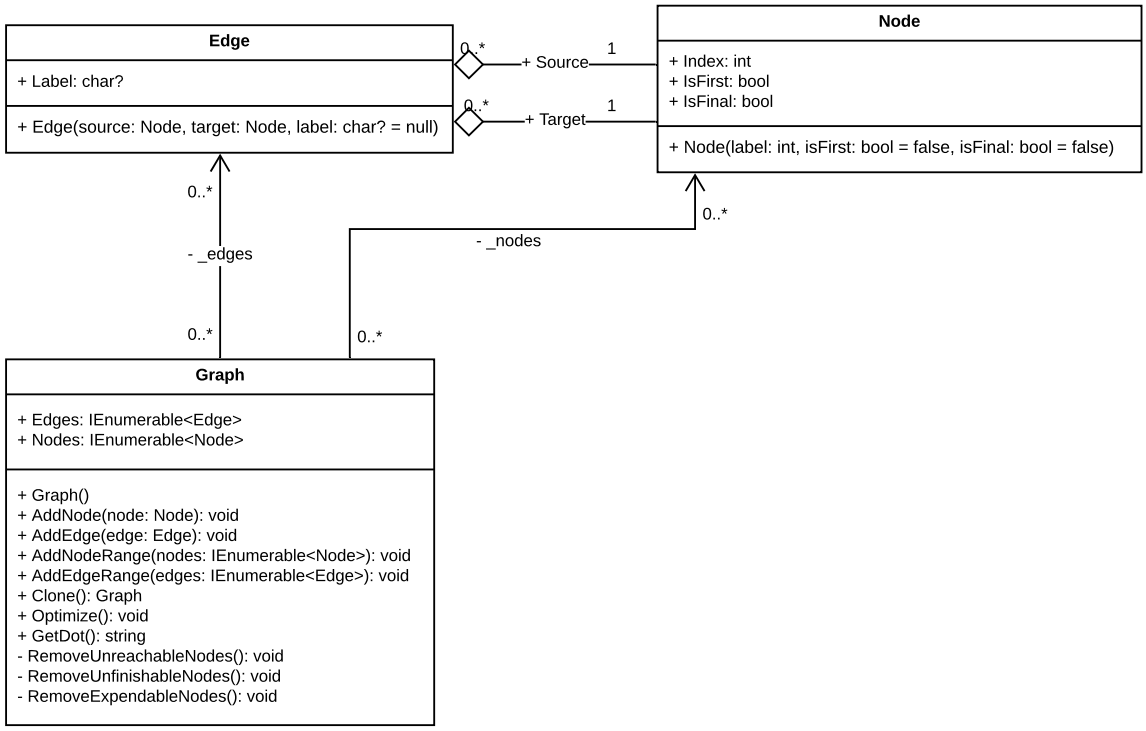
6 Architektura

Tento aplikace nabízí webové rozhraní. Podle účelu byla však rozdělena do následujích tří knihoven, které projekt typu ASP.NET core využívá: Automaton (6.1), SimpleRegEx (6.2), Definitions (6.3). Ty poskytují metody pro práci s regulárními výrazy a grafy (v našem případě konečnými automaty).

6.1 Knihovna Automaton

Knihovna automaton implementuje strukturu grafu, ke kterému se můžeme chovat libovolně, tzn. můžeme v něm vytvářet jakékoliv grafy. Za správnost jednotlivých grafů (tj. splňují podmínky NKA) zodpovídají samotná pravidla (viz podsekci 6.3.3).

Knihovna obsahuje 3 třídy: „`Graph`“, „`Edge`“ a „`Node`“. Její třídní diagram je k dispozici na obrázku 5.



Obrázek 5: Třídní diagram knihovny Automaton

6.1.1 Třída Graph

Třída Graph je hlavní třídou, která uchovává celou strukturu grafu. Implementuje metody pro přidávání uzlů (`AddNode`, `AddNodeRange`) a hran (`AddEdge`, `AddEdgeRange`) nebo pro optimali-

zaci (`Optimize`), generování reprezentace grafu v jazyce DOT (`GetDot`) a klonování struktury (`Clone`).

Do objektu této třídy mohou být přidávány samostatně jak uzly, tak hrany, aby mohly vznikat grafy, obsahující uzly, jež se nenachází v žádné hraně ani jako výchozí ani cílový uzel. Může být také přidána hrana, jejíž výchozí nebo cílový uzel ještě nebyl přidán. V takovém případě se uzly sice vykreslí, ale nenalézají se v kolekci uzlů grafu.

6.1.2 Třída Edge

Třída Edge reprezentuje hrany v grafu. Udává její výchozí a cílový uzel a znak, který musí být na vstupu, aby se po ní mohlo přejít.

6.1.3 Třída Node

Třída Node reprezentuje uzel grafu (neboli stav automatu). Neimplementuje žádné metody, pouze ukládá index (neboli označení) uzlu a informace, zda je vstupním a/nebo přijímacím uzlem. Index nemusí být unikátní. Slouží pouze k jednodušší orientaci v grafu.

6.2 Knihovna SimpleRegEx

Knihovna SimpleRegEx obstarává tvorbu regulárního výrazu. Ten je ukládán v upravené stromové struktuře. Obsahuje pouze jednu veřejnou třídu (`RegEx`). Používá ale další 2 interní třídy (`Tree`, `TreeNode`) a jedno vyjmenování (`TreeNodeType`), které lze používat pouze v rámci knihovny.

Třídní diagram této knihovny je na obrázku 6.

6.2.1 Třída RegEx

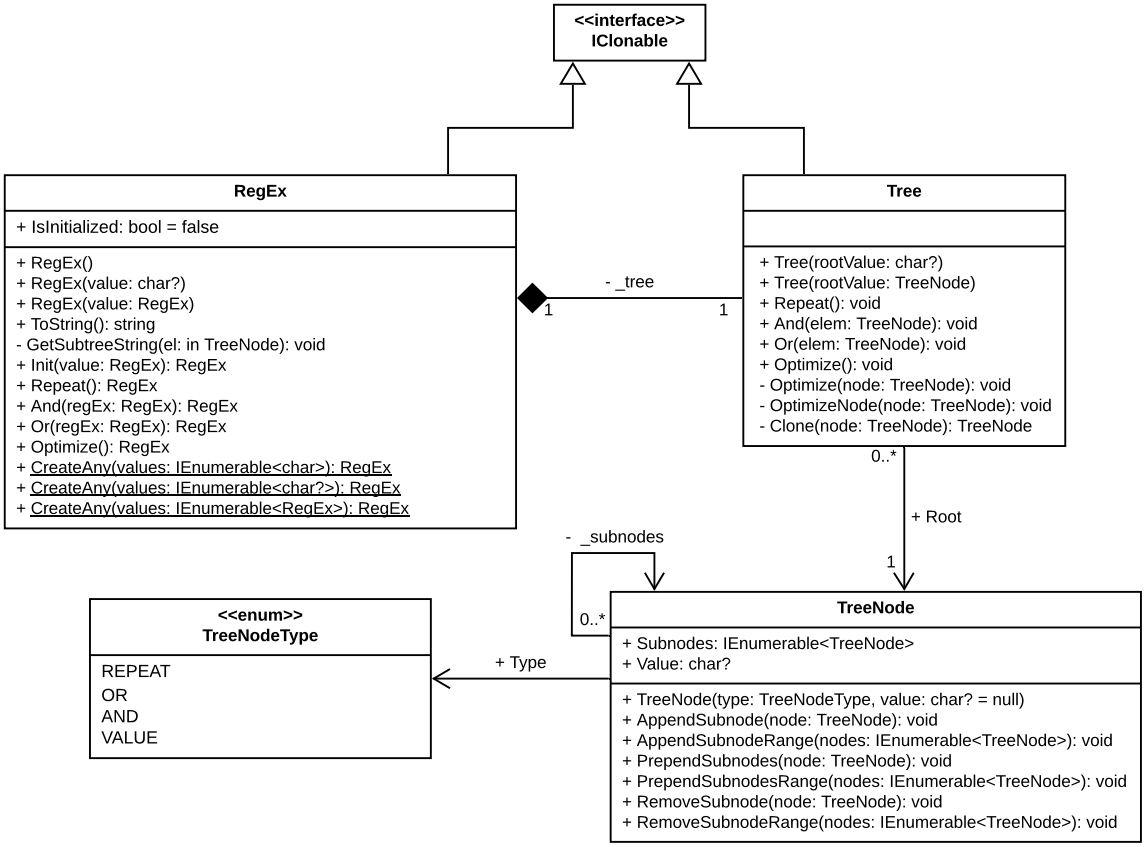
Zde se skládá samotný regulární výraz do stromové struktury. Nabízí k tomu metody `Repeat`, `And` a `Or`. Aby je však šlo použít, musí být výraz inicializovaný, tzn. musí mít zadaný kořen. Ten lze specifikovat buď pomocí metody `Init` nebo jako parametr konstruktoru. Všechny tyto metody používají návrhový vzor řetězcového volání (*Fluent Interface*)[3].

Dalšími funkcemi, které třída nabízí, jsou optimalizace (metoda `Optimize`), klonování struktury (`Clone`) nebo metoda `ToString`, která umí ze sestaveného stromu vytvořit výslednou textovou reprezentaci regulárního výrazu.

Třída nabízí i statickou metodu `CreateAny` pro vytvoření regulárního výrazu, přijímacího jakýkoliv znak ve specifikované množině regulárních výrazů. Metoda má 3 přetížení. Přijímá kolekci znaků (`char`), znaků, které mohou nabývat hodnoty `null` (`char?`), případně samotných regulárních výrazů. Například pro kolekci znaků {a, b, c} vznikne výraz „$(a + b + c)$“.

Na regulární výraz lze implicitně převádět znaky (`char`).

[3]Objekt obsahující takovou metodu je na jejím konci vrácen. Mohou tak vznikat konstrukce jako například „`new RegEx('a').And(new RegEx('b')).Repeat()`“, čímž vznikne regulární výraz $(a \cdot b)^*$.



Obrázek 6: Třídní diagram knihovny SimpleRegEx

6.2.2 Třída Tree

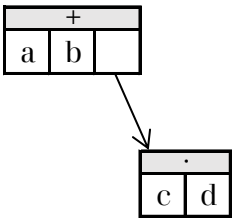
Tato třída ukládá strukturu regulárního výrazu. Implementuje některé metody, podobné těm ve třídě RegEx, jako `And`, `Or`, `Repeat`, `Optimize` nebo `Clone`.

Na kořen celého stromu odkazuje vlastnost `Root`.

6.2.3 Třída TreeNode

Strom se skládá z uzlů. Každý má svůj typ (viz podsekce 6.2.4), odkaz na rodiče (`Parent`; nabývá hodnoty `null`, pokud se jedná o kořen stromu), poduzly (`Subnodes`; pro uzly typu `VALUE` je `null`) a hodnotu (`Value`; pro všechny typy uzlů, kromě `VALUE`, nabývá hodnoty `null`).

Poduzlů může být libovolné množství. Shlukují se podle společných operátorů. Stromová struktura je tedy jednodušší, než při vytváření klasického binárního stromu a lze na ni jednodušeji provádět různé optimalizace. Příklad takového stromu pro výraz „$a+b+(c\cdot d)$“ je na obrázku 7.



Obrázek 7: Příklad stromu, vytvořeného pro regulární výraz $a + b + (c \cdot d)$

6.2.4 Vyjmenování TreeNodeType

Každý uzel stromu má svůj typ. Vyjmenování `TreeNodeType` je definuje. Podle tohoto typu jsou poté, při sestavování textové reprezentace, určeny vstahy mezi poduzly. Může nabývat hodnot `REPEAT` (opakování podslova), `OR` (disjunkce podslov), `AND` (konjunkce podslov) nebo `VALUE` (reprezentuje konkrétní hodnoty).

6.3 Knihovna Definitions

Tato knihovna poskytuje například samotná pravidla, která lze použít pro generování NKA nebo regulárního výrazu, a abecedu jazyka. Její třídí diagram je k nahlédnutí na obrázku 8.

6.3.1 Třída FormalLanguage

Řídící třídou je zde `FormalLanguage`. Ta ukládá výše uvedené části formální definice jazyka (pravidla, abeceda). Nabízí také rozhraní pro práci s nima. Implementuje metody `GetAutomaton` vracející textovou reprezentaci NKA v jazyce DOT, `GetRegEx` ke zjištění regulárního výrazu jazyka a metodu `IsValid`, která ověřuje správnost jazyka.

Podmínkami správnosti jazyka jsou neprázdná abeceda a správnosti jednotlivých použitých pravidel, jejichž ověření si spravují pravidla samotná.

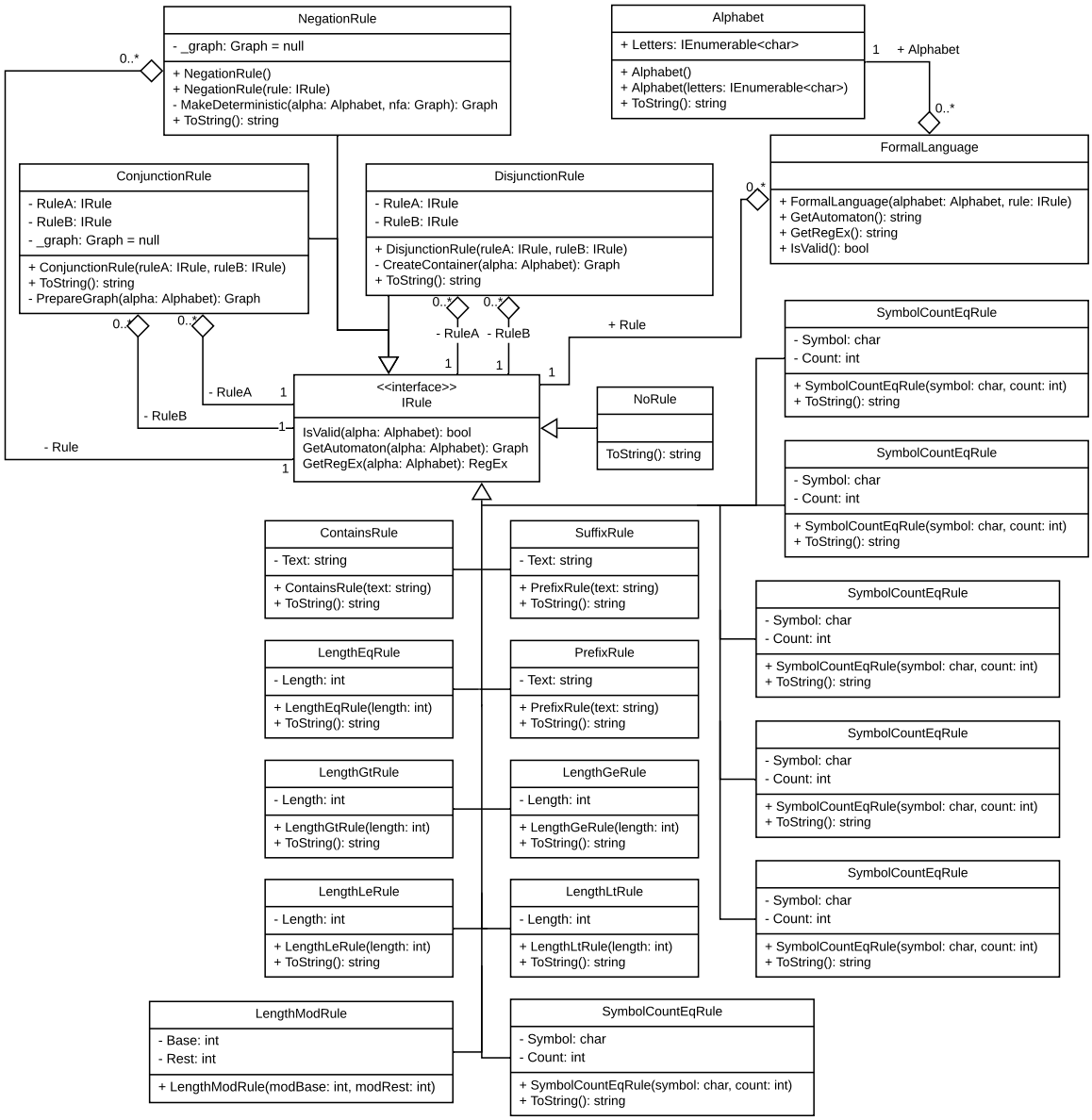
6.3.2 Třída Alphabet

Objekt této třídy reprezentuje abecedu jazyka. Ta se skládá z libovolného počtu znaků. Třída nenabízí žádné metody, pouze vlastnost `Letters`, tedy seznam znaků v abecedě.

6.3.3 Rozhraní IRule

Toto rozhraní by měla implementovat všechna pravidla. Nabízí metody pro tvorbu regulárního výrazu (`GetRegEx`), konečného automatu (`GetAutomaton`) a ověření, zda je pravidlo validní (`IsValid`).

Pro každou podmínku na požadovaný jazyk existuje speciální třída, která ji řeší. Podmínky (neboli pravidla) můžeme rozdělit na 2 druhy:



Obrázek 8: Třídní diagram knihovny Definitions

- **Atomická pravidla**: Nacházejí se ve jmenném prostoru Definitions.UsableRules a označují základní pravidla jako například „`prefix`“ nebo „`suffix`“.
- **Logické spojky**: Ty jsou ve jmenném prostoru Definitions.BaseRules a slouží ke spojování atomických pravidel do složitějších struktur pomocí logických operací. Výjimku zde tvoří třída `NoRule`, která je použita, pokud nebyly specifikovány žádné podmínky jazyka.

6.4 Projekt NFA_RegEx_Generator

Tento projekt nabízí webové uživatelské rozhraní. To je implementováno pomocí frameworku ASP.NET core verze 3.1. Veškerý text je psán anglicky.

Projekt obsahuje 3 kontrolery. `HomeController` pouze zobrazuje domovskou stránku se základními informacemi, jak program funguje a co je například formální jazyk nebo konečný automat.

`InstructionsController` zobrazuje konkrétní informace například o tom, v jakém formátu se má zadávat abeceda nebo podrobnosti o jednotlivých pravidlech.

`GeneratorController` řídí samotné výpočty konečných automatů, regulárních výrazů, ale také zpracování zadaných pravidel a abecedy.

Projekt si také sám zpracovává například řetězce specifikující abecedu a pravidla. Nástroje k tomu potřebné jsou dostupné ve jmenném prostoru `NFA_RegEx_Generator.Utilities`.

6.4.1 Výjimka ParserException

Tato výjimka je vyhazována vždy, když při zapracovávání zadaného výrazu nastane chyba. Její konstruktor přijímá jako parametry pozici, na které nastala chyba a zprávu. Oba parametry jsou díky několika přetížením nepovinné.

6.4.2 Zpracování abecedy

Abeceda se zadává do textového pole jako kolekce alfa-numerických znaků oddělených čárkami. Všechny prázdné znaky jsou ignorovány.

Parser zpracovávající abecedu (`AlphabetParser`) se nachází ve jmenném prostoru `NFA_RegEx_Generator.Utilities`. Využívá návrhového vzoru singleton a k jeho instanci lze přistupovat pomocí statické vlastnosti `Instance`.

Poskytuje pouze jednu veřejnou metodu a to metodu `Parse`, která bere jako svůj parametr zadaný řetězec. Pokud nějaký zadaný znak nemá délku rovnu jedné nebo není žádný znak zadán, je vyhozena výjimka `ParserException`.

6.4.3 Pravidla

K dispozici jsou následující pravidla. Jejich parametry musí být oddělené čárkou. Čísla, znaky ani řetězce nemusí být označeny uvozovkami. Prázdné znaky jsou ignorovány.

- **prefix**
    - **Třída**: `PrefixRule`
    - **Parametr**: Řetězec, jehož všechny znaky jsou v abecedě
    - **Popis**: Specifikované podslovo se musí nacházet na začátku každého slova
    - **Formát**: `prefix(<podslovo>)`

- **Příklad**: `prefix(ab)` nad abecedou Σ={a, b}

- **contains**
  - **Třída**: `ContainsRule`
  - **Parametr**: Řetězec, jehož všechny znaky jsou v abecedě
  - **Popis**: Každé slovo, patřící do jazyka musí obsahovat dané podslovo
  - **Formát**: `contains(<podslovo>)`
  - **Příklad**: `contains(ooq)` nad abecedou Σ={o, p, q}

- **suffix**
  - **Třída**: `SuffixRule`
  - **Parametr**: Řetězec, jehož všechny znaky jsou v abecedě
  - **Popis**: Specifikuje podslovo, které se musí nacházet na konci každého slova jazyka
  - **Formát**: `suffix(<podslovo>)`
  - **Příklad**: `suffix(01)` nad abecedou Σ={0, 1}

- **length**
  - **Třídy**: `LengthEqRule`, `LegnthLtRule`, `LengthLeRule`, `LengthGeRule`, `LengthGtRule`, `LengthModRule`
  - **Parametry**: Prvním parametrem je operátor (viz 6.4.4). Pokud byl zadán operátor „mod“, pak délka slova modulo základ modula (základ modula je 2. parametr) se musí rovnat zbytku po celočíselném dělení (3. parametr).

    Při použití jakéhokoliv jiného operátoru bude opravdová délka slova se zadanou délkou (2. parametr) ve vstahu definovaném operátorem.
  - **Formát**: `length(<operátor>, <délka slova>)` nebo `length(mod, <základ modula>, <zbytek po dělení>)`
  - **Příklad**: `length(>, 2)` nad abecedou $\Sigma = \{a\}$ nebo `length(mod, 3, 2)` nad abecedou $\Sigma = \{a, b\}$

- **symCount**
  - **Třídy**: `SymbolCountEqRule`, `SymbolCountLtRule`, `SymbolCountLeRule`, `SymbolCountGeRule`, `SymbolCountGtRule`, `SymbolCountModRule`
  - **Parametry**: Podobně jako u pravidla „`length`“ je prvním parametrem operátor. Pokud je jím „mod“, tak počet zadaných znaků (2. parametr - jedno písmeno z abecedy) modulo základ modula (3. parametr) se musí rovnat zbytku po celočíselném dělení (4. parametr).

Pokud ja zadán jiný operátor, pak musí být skutečný počet zadaných znaků (znak - 2. parametr) s uvedeným počtem (3. operátor) ve vstahu definovaném operátorem.

- **Formát**: `symCount(<operátor>, <znak>, <počet výskytů znaku>)` nebo `symCount(mod, <znak>, <základ modula>, <zbytek po dělení>)`
- **Příklad**: `symCount(<, x, 3)` nad abecedou $\Sigma = \{x, y\}$, případně pro operaci modulo: `symCount(mod, n, 5, 1)` nad abecedou $\Sigma = \{m, n\}$

Pravidla mohou být také spojovány pomocí logických operátorů:

- `&&` (Průnik - Konjunkce)
  - **Třída**: `ConjunctionRule`
  - **Popis**: Před i za tímto operátorem musí být specifikována pravidla. Obě poté musí platit
  - **Formát**: `<ruleA> && <ruleB>`
  - **Příklad**: `prefix(ab) && suffix(ba)` nad abecedou Σ={a, b}
- `||` (Sjednocení - Disjunkce)
  - **Třída**: `DisjunctionRule`
  - **Popis**: Před i za tímto operátorem musí být specifikována pravidla. Alespoň jedno pravidlo pak musí platit
  - **Formát**: `<ruleA> || <ruleB>`
  - **Příklad**: `contains(xy) || length(=, 3)` nad abecedou Σ={x, y, z}
- `!` (Negace)
  - **Třída**: `NegationRule`
  - **Popis**: Za tímto operátorem musí být specifikováno pravidlo. Pak platí opak tohoto pravidla
  - **Formát**: `!<rule>`
  - **Příklad**: `!prefix(m)` nad abecedou Σ={m, n}

Výrazy lze i závorkovat a tak měnit jejich prioritu. K uzávorkovanému výrazu se pak můžeme chovat jako k pravidlu, tzn. můžeme nad ním provádět například negaci nebo jej spojit s jiným pravidlem pomocí logického operátoru. Závorky je možno vynechávat podle konvence pro vypouštění závorek.

Posledním pravidlem je „žádné pravidlo“. Třída je `NoRule` a její instance je vytvořena pouze, když nejsou specifikovány žádné podmínky. Poté každé slovo nad zadanou abecedou do jazyka patří.

6.4.4 Operátory

- `<` (Menší než): Počet (myšleno počet znaků ve slově celkem nebo četnost zadaného znaku) musí být menší, než číslo uvedené v parametru
- `<=` (Menší nebo rovno): Počet musí být menší nebo roven zadanému číslu
- `=` (Rovno): Počet musí být roven zadanému číslu
- `>=` (Větší nebo rovno): Počet musí být větší nebo roven zadanému číslu
- `>` (Menší nebo rovno): Počet musí být větší než zadané číslo
- `mod` (Modulo): Počet modulo uvedený základ musí být rovno zadanému zbytku po dělení

6.5 Zpracování pravidel

Zadaný řetězec pravidel je zpracováván ve třídě `RulesParser`, nacházející se ve jmenném prostoru `NFA_RegEx_Generator.Utilities`. Parser je implementován jako singleton a k jeho instanci se dá přistoupit pomocí statické vlastnosti `Instance`. Nabízí pouze jednu veřejnou metodu `Parse`. Ta za pomoci rekurzivního sestupu zadaný řetězec zpracuje a vrátí výsledné pravidlo. To je po zpracování vždy právě jedno (byť složené z více atomických pravidel logickými spojkami).

6.5.1 Gramatika

`RulerParser` obsahuje soukromou třídu `Grammar`. Ta implementuje bezkontextovou gramatiku, definovanou následovně:

$$
\begin{aligned}
&\mathrm{A} \rightarrow \mathrm{C\ B} \\
&\mathrm{B} \rightarrow \vee\ \mathrm{A} \mid \varepsilon \\
&\mathrm{C} \rightarrow \mathrm{E\ D} \\
&\mathrm{D} \rightarrow \wedge\ \mathrm{C} \mid \varepsilon \\
&\mathrm{E} \rightarrow \texttt{(}\ \mathrm{A}\ \texttt{)} \mid \texttt{!}\ \mathrm{E} \mid \texttt{<Pravidlo>}
\end{aligned}
$$

Gramatika zajišťuje priority operátorů. Zpracování operátorů se stejnou prioritou je zprava. V této třídě se nachází také další 4 vnořené třídy a jedno vyjmenování.

6.5.2 Třída Scanner

Třída `Scanner` umí rozdělit řetězec na jednotlivé tokeny. Jeho konstruktor přijímá jako parametr řetězec, který má objekt zpracovávat. Pomocí metody `NextToken` mohou být jednotlivé tokeny postupně odebírány.

6.5.3 Vyjmenování TokenType

Každý token, který třída `Scanner` vytvoří, má nějaký typ.

- `EOL`: Označuje konec řetězce. Poslední token vrácený medotou `NextToken` třídy `Scanner`.
- `NEG`: Byl přečten symbol negace („`!`“).
- `CON`: Znak konjunkce („`&&`“)
- `DIS`: Znak disjunkce („`||`“)
- `RULE`: Token obsahuje data atomického pravidla. Pravidlo však ještě není rozparsováno. O to se stará až třída `Tree`.
- `LBRA`: Levá závorka
- `RBRA`: Pravá závorka

6.5.4 Třída Token

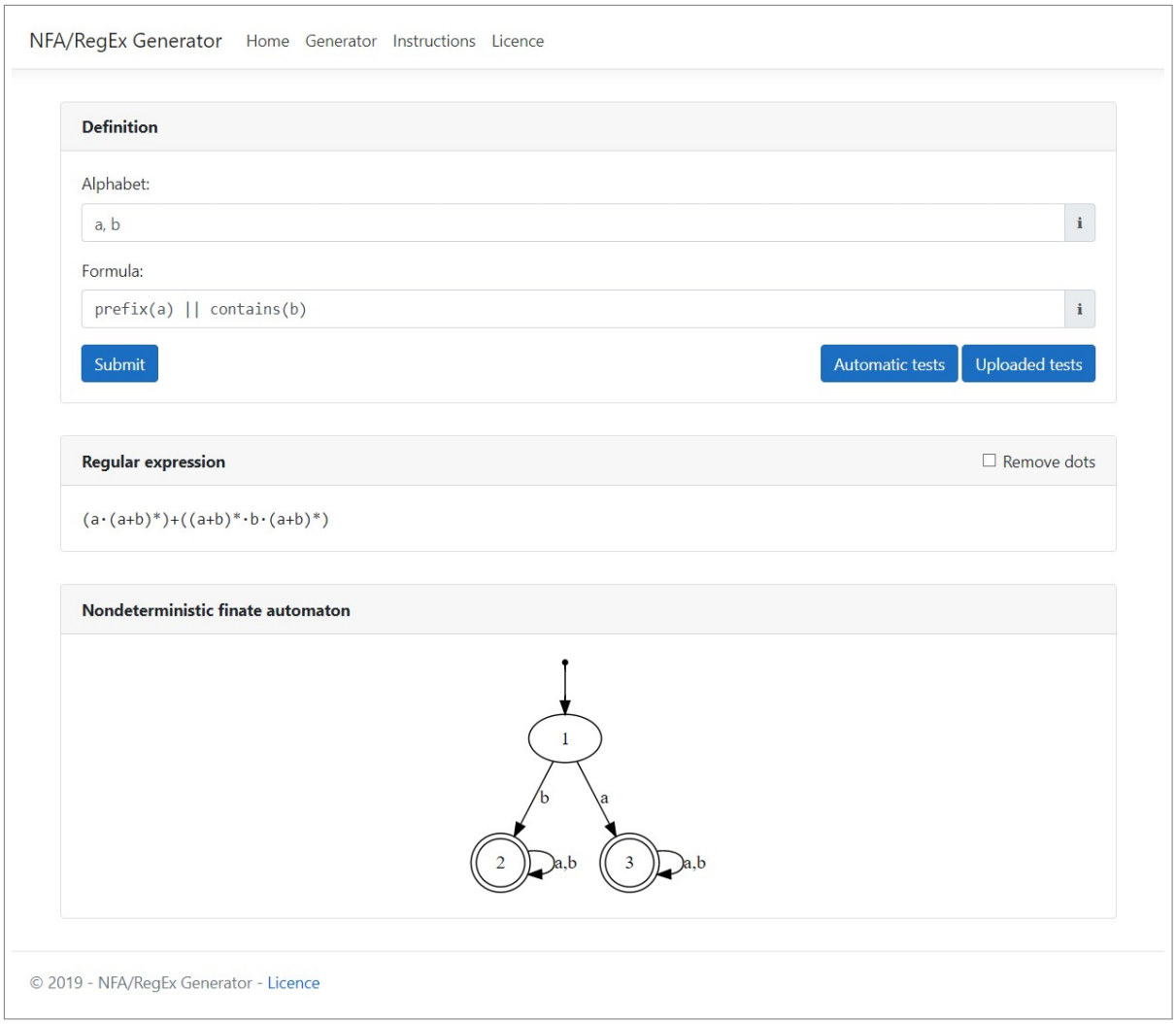
Tato třída ukládá typ (`Type`) a hodnotu (`Value`) tokenu. Obě vlastnosti jsou určeny pouze pro čtení a lze je inicializovat pomocí konstruktoru. Hodnotu však musí mít pouze tokeny typu `RULE`. U všech ostatních typů je ignorována.

6.5.5 Třída Tree

Třída `Tree` reprezentuje strukturu syntaktického stromu. Je do něj možno přidávat nové uzly (metoda `AddElement`), uzlům specifikat hodnotu (`AddValue`) a posouvat se ve stromu o úroveň výše (`MoveBack`).

Třída také vytváří samotná pravidla. O to se strará metoda `GetRule`, která vrátí výsledné pravidlo. Využívá k tomu několik soukromých metod. Jednou z nich je `ParseUsableRule`, která zpracovává atomická pravidla.

6.5.5.1 Třída TreeElement Objekt třídy `TreeElement` reprezentuje jeden uzel v syntaktickém stromu. Je v něm uložen token jako jeho hodnota (`Value`) a odkazy na větve z něho vycházející (`Left`, `Right`). Pro možnost vrátit se ve stromu o úroveň výš, poskytuje objekt také odkaz na svého rodiče (`Parent`). Poslední vlastností objektu této třídy je pozice v řetězci, kde byl token přečten. Pokud při zpracovávaní nastane chyba, je vyhozena výjimka `ParserException` (viz 6.4.1) s parametrem `position` a může tak být později zobrazeno, kde přesně je v řetězci chyba.



Obrázek 9: Ukázka webového rozhraní aplikace

6.6 Uživatelské rozhraní

Projekt nabízí webové rozhraní, implementované pomocí ASP.NET Core 3.1 za pomoci návrhového vzoru MVC. Její náhled a příklad použití jsou znázorněny na obrázku 9.

V horní části aplikace se nachází 4 záložky. První záložka „Home“ je odkazem na úvodní stránku, kde jsou napsány základní informace o aplikaci, jak ji používat a stručná teorie o konečných automatech a regulárních výrazech.

Pod třetí záložkou můžeme nalézt podrobné instrukce, jak aplikaci používat. Tzn. jak zadávat abecedu a pravidla. Ta tam jsou také vypsána s krátkým popisem, co dané pravidlo znamená a formát, v jakém musí být zadáno.

Hlavní stránkou aplikace je „Generator“. Zde se nachází rozhraní pro zadávání abecedy a pravidla. U každého z těchto textových polí je tlačítko, které zobrazí rychlou nápovědu.

Pokud jsou zadané vstupy ve správném formátu, ve spodní části se po stisknutí tlačítka

„Submit“ zobrazí vygenerované informace. Regulární výraz je vypsán jako klasický řetězec. U konkatenace lze odstranit tečky, aby se dalo ve výrazu lépe orientovat. Pod ním je vykreslen konečný automat.

6.6.1 Generované příklady

Na stránce generátoru si lze také nechat náhodně sestavit vstupy k odtestování funkčnosti aplikace. Ty se na stránce zobrazí po rozkliknutí tlačítka „*Generated examples*“. Dále může uživatel vybrat, jaké pravidlo chce testovat. Literály v pravidlech jsou definovány náhodně, stejně jako operátory u pravidel, které je vyžadují, nebo abeceda. Ta může obsahovat 1 až 5 znaků. Vytvořený výraz lze upravit a znovu spustit jako uživatelem zadaný jazyk.

6.6.2 Nahrané příklady

Při rozkliknutí tlačítka „*Uploaded examples*“ se zobrazí testy nahrané administrátorem. O tuto funkcionalitu se stará třída `TestReader`.

V souboru `appsettings.json` projektu je specifikována cesta do složky s testy. V objektu třídy je nastavený `FileSystemWatcher`, který sleduje aktivitu XML souborů ve složce. Pokud je nějaký přidán nebo změněn, zkusí jej zpracovat. Pokud soubor splňuje podmínky správnosti, zobrazí se v seznamu nahraných testů.

Pokud je soubor přejmenován, test se znovu nezpracovává, pouze se změní jeho název.

Při smazání souboru, je odstraněn i test.

6.6.3 Formát nahraného testu

Soubor testu musí být formátu XML a jeho kořenový uzel je `FormalLanguage`. Dalšími povinnými poduzly jsou `Alphabet`, pro specifikaci abecedy a `Formula` pro zadání pravidla. Nepovinnými poduzly jsou `RegEx` a `Graph`, které slouží pro vytvoření konkrétního regulárního výrazu a grafu.

Pokud je zadán jak graf, tak regulární výraz, zadané hodnoty se pak nezpracovávají a může být jako abeceda i pravidlo uveden libovolný řetězec. Je to z toho důvodu, že například učitel může chtít v aplikaci demonstrovat pravidlo, které aplikace neimplementuje. Pokud však nějaký z těchto poduzlů chybí, aplikace se jej snaží doplnit podle zadaných hodnot. Nejsou-li však ve formátu, v jakém je parser požaduje, je zobrazena chyba. Zadaná část se vykreslí i v tomto případě.

Jak již bylo uvedeno, abeceda i pravidlo se zadává jako jednoduchý řetězec. To stejné platí i pro regulární výraz. V grafu však musí být specifikovány jak uzly, tak hrany. K tomu slouží poduzly `Node` a `Edge`.

Poduzel `Node` může obsahovat atributy, zda je vstupním (`First`) a/nebo přijímacím (`Final`) - hodnota těchto atributů musí být nastavena na „`true`“, jinak jsou ignorovány. V hodnotě poduzlu musí být uvedeno jeho označení (zde může být zadán i libovolný řetězec).

Hrana můsí mít uvedeny atributy `Source` (výchozí uzel) a `Target` (cílový uzel). Hodnotou uzlu je řetězec, který se objeví u hrany, primárně určený na vyjmenování znaků, přes které automat hranou přechází.

```
<FormalLanguage>
    <Alphabet>a, b</Alphabet>
    <Formula>prefix(ab)</Formula>
    <RegEx>foo</RegEx>
    <Graph>
        <Node First="true">a</Node>
        <Node>b</Node>
        <Node>c</Node>
        <Node Final="true">d</Node>
        <Edge Source="a" Target="b">testAB</Edge>
        <Edge Source="b" Target="c">testBC</Edge>
        <Edge Source="c" Target="d">testCD</Edge>
    </Graph>
</FormalLanguage>
```

Výpis 2: Příklad nahraného testu

7 Použité algoritmy

7.1 Tvorba NKA

Každé pravidlo implementující rozhraní IRule musí, mimo jiné, implementovat i metodu pro generování konečného automatu - `GetAutomaton`. Každá třída využívá pro toto generování vlastní algoritmus. Třídy implementující pravidla `length` a `symCount` pouze pro různé operátory (např. `LengthEqRule` a `LengthGtRule`) mají tyto algoritmy často podobné. Proto u těchto tříd budou v sekcích 7.1.4 a 7.1.5 popsány postupy pouze pro operátor „`=`“ (rovná se).

7.1.1 PrefixRule

K vytvoření konečného automatu je ve třídě `PrefixRule` využit cyklus, který proběhne tolikrát, kolik je délka zadaného podslova. Na jeho začátku je vytvořen nový uzel, který je ke konci iterace uložen do proměnné `lastNode`, aby byl dostupný i v příští iteraci. Pokud je cyklus procházen poprvé, nově vytvořený uzel je označen jako počáteční. V opačném případě je vytvořena také hrana mezi `lastNode` a novým uzlem.

V poslední iteraci je nový uzel označen jako přijímací a pro každý znak v abecedě je vytvořena smyčka do něj samotného.

7.1.2 ContainsRule

Postup vytváření NKA pro pravidlo `contains` je možno rozdělit na 2 části, které běží za sebou v cyklu pro každý znak v zadaném podslovu.

Nejdříve se vytvoří nový uzel a hrana, která do něj povede ze stavu, vytvořeném v minulé iteraci (při první iteraci je použit uzel, vytvořený před začátkem cyklu, který je později označen jako počáteční). Tato hrana bude přijímat znak, který je v zadaném podslově další v pořadí. Všechny nově vytvořené uzly se ukládají do senzamu `nodes`.

Další část určuje, kam povedou hrany pro ostatní znaky z uzlu vytvořeném v minulé iteraci (případě počátečního). V cyklu od 1 po `i` včetně, kde `i` je kolikátá iterace vnějšího cyklu probíhá mínus jedna, hledáme podslovo zadaného slova. To začíná na jeho indexu 1 a končí indexem `i`. K němu je ještě přidán právě zpracovávaný znak. Pokud tímto podslovem začíná zadané slovo, pak délka podslova je rovna indexu uzlu v `nodes`, do kterého má hrana směřovat. Pokud podslovo není na začátku zadaného slova, odebere se z něj první znak a postup se opakuje. Pokud žádné takoto vytvořené podslovo není nalezeno, indexem je nula, což znamená, že hrana povede do počátečního uzlu.

Po skončení vnějšího cyklu je první uzel v `nodes` označen jako počáteční a poslední jako přijímací. Pro každý znak v abecedě jsou vytvořeny smyčky k přijímacímu uzlu do něj samotného.

Implementace této metody je dostupná ve výpisu 3.

7.1.3 SuffixRule

Třída `SuffixRule` používá velice podobný postup, jako `ContainsRule`. Algoritmus, popsaný v druhé části, je pouze použit i pro poslední vytvořený, přijímací uzel. Po skončení vnějšího cyklu se také nevytváří smyčky z přijímacího uzlu do něj samotného.

```
public Graph GetAutomaton(Alphabet alpha) {
   var g = new Graph();
   var nodes = new List<Node>();
   nodes.Add(new Node(1));
   g.AddNode(nodes[0]);

   for (var i = 0; i < Text.Length; i++) {
      nodes.Add(new Node(i + 2)); // Vytváří se "dopředu".
      g.AddNode(nodes[i + 1]);
      g.AddEdge(new Edge(nodes[i], nodes[i + 1], Text[i]));

      foreach (var l in alpha.Letters.Where(l => l != Text[i])) {
         int idx = 0;
         for (var j = 1; j <= i; j++) {
            if (Text.StartsWith(Text[j..i] + l)) {
               idx = Text[j..i].Length + 1;
               break;
            }
         }
         g.AddEdge(new Edge(nodes[i], nodes[idx], l));
      }
   }

   nodes[0].IsFirst = true;
   nodes[^1].IsFinal = true;
   foreach (var l in alpha.Letters)
      g.AddEdge(new Edge(nodes[^1], nodes[^1], l));
   return g;
}
```

Výpis 3: Metoda GetAutomaton třídy ContainsRule

7.1.4 LenghtEqRule

Algoritmus pro tvorbu konečného automatu je u tohoto pravidla velice jednoduchý. Vytvoří se n+1 vrcholů, kde n je požadovaná délka slova, mezi kterými jsou přechody pro každý znak v abecedě. První vytvořený uzel je označen jako počáteční a poslední jako přijímací.

7.1.5 SymbolCountEqRule

Opět je vytvořeno n+1 uzlů, kde n je požadovaná četnost znaku ve slově. Každý obsahuje smyčku sám do sebe pro všechny znaky abecedy, kromě zadaného znaku a mezi nima jsou přechody pro zadaný znak. První vytvořený uzel je označen jako počáteční a poslední jako přijímací.

7.1.6 ConjunctionRule

V tomto algoritmu je použit postup pro převod NKA na DKA. Protože byl použit v programu vícekrát, tak je popsán v samostatné sekci 7.1.9.

Spojované grafy jsou vloženy do jednoho nového (jsou do něj vloženy všechny uzly a hrany, jak prvního, tak druhého grafu). Poté je aplikován postup převodu NKA na DKA, pouze jsou označeny jiné stavy jako přijímací. Těmi jsou takové stavy, které jsou přijímací v obou původních grafech.

7.1.7 DisjunctionRule

Tady se také vytvoří nový graf, do kterého jsou vloženy oba původní a aplikuje se na něj převod NKA na DKA, tentokrát tak, jak je popsán v 7.1.9.

7.1.8 NegationRule

Opět musíme aplikovat převod NKA na DKA, ale tentokrát na původní graf (pravidlo vyžaduje pouze jeden). Poté změníme všechny přijímací stavy na nepřijímací a naopak nepřijímací stavy na přijímací.

7.1.9 Převod NKA na DKA

Převod NKA na DKA se v této práci používá například při vytváření konečného automatu u konjunktinvího pravidla nebo negace. Je při něm použit stromový algoritmus.

1. Vytvoříme kořen stromu spojením všech možných počátečních stavů do jedné množiny.
2. Vytvoříme novou vrstvu stromu takovým způsobem, že pro každý stav z příslušné množiny předchozí vrstvy, vytvoříme novou množinu všech stavů, které jsou stavem následujícím po přechodu příslušnou hranou.
3. Pokud se nová množina stavů v automatu již nachází, nevytváří se nový uzel, ale vytvoří se přechod do takového, již existujícího, stavu.
4. Opakujeme kroky číslo 2 a 3 tak dlouho, dokud se vytváří nové množiny stavů, které se ještě ve stromové hierarchii neobjevily.
5. Označíme všechny podmnožiny, které obsahují přijímací stav původního automatu. Tyto množiny se stanou přijímací stavy nového automatu.

6. Jakmile je strom hotov, přejmenujeme jednotlivé podmnožiny na nové označení nového automatu.

7. Nyní můžeme vytvořit stavový diagram nově vzniklého deterministického konečného automatu.

[10]

7.2 Optimalizace NKA

V praktické části této práce je optimalizace grafu konečeného automatu implementována pomocí 3 algoritmů pro eliminaci stavů.

7.2.1 Odstranění mrtvých stavů

První algoritmus odstraní z grafu stavy, ze kterých se nedá, pomocí hran, dostat do žádného přijímacího stavu.

V cyklu jsou procházeny všechny přijímací stavy. Každý tento stav je uložen do zásobníku. Druhý, vnořený cyklus prochází zásobník, kam jsou postupně přidávány všechny výchozí stavy hran, směřující do aktuálně zpracovávaného stavu. Každý takto zpracovaný stav je přidán do seznamu navštívených stavů, čímž se označí, aby později nebyl smazán.

Po skončení obou cyklů jsou odstraněny všechny stavy, které nejsou v seznamu navštívených stavů a také hrany do nich směřující. Část zdrojového kódu, implementujícího tento algoritmus je ve výpisu 4.

7.2.2 Odstranění nedosažitelných stavů

Tento algoritmus odstraňuje nedosažitelné uzly popsané v definici 4. Postup je velmi podobný jako u přechozího algoritmu pro odstranění mrtvých stavů. Vnější cyklus však prochází počáteční stavy a do zásobníku se přidávají cílové stavy hran, vycházejících z právě zpracovávaného stavu. Zdrojový kód, implementující tento algoritmus je ve výpisu 5.

Definice 4 *Stav q konečného automatu A = (Q, Σ, δ, I, F) je* ***dosažitelný*** *pokud existuje nějaké slovo w takové, že* $I \xrightarrow{w} q$.

V opačném případě stav nazýváme ***nedosažitelný****. [11]*

7.2.3 Redukování izolovaných skupin přijímacích stavů

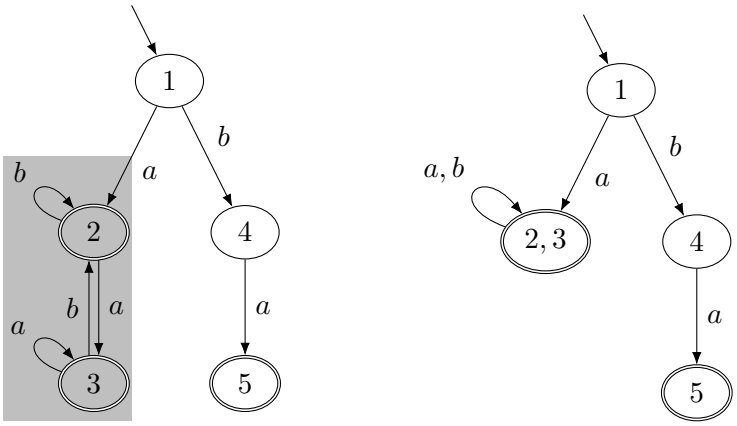
Posledním implementovaný algoritmus pro snížení počtu stavů v NKA shlukuje izolované skupiny přijímacích uzlů, které mají stejné, z nich vycházející hrany (myšleno stejné znaky, pomocí kterých se může hranou přejít). Izolovaností zde chápeme, že ze skupiny nevede žádná hrana. Tyto uzly jsou všechny nahrazeny jedním přijímacím uzlem se smyčkami pro každý znak, který se ve skupině nacházel. Příklad takovéto situace je vyobrazen na obrázku 10.

```
var visited = new List<Node>(); // seznam navštívených uzlů
foreach (var node in _nodes.Where(n => n.IsFinal)) {
   var stack = new Stack<Node>(); // zásobník
   stack.Push(node);
   while (stack.TryPop(out var n)) {
      if (visited.Contains(n))
         continue;
      visited.Add(n);
      var sources = _edges // výchozí uzly všech hran, kde "n" je cílový uzel
         .Where(e => e.Target == n)
         .Select(e => e.Source);
      foreach (var s in sources)
            stack.Push(s);
   }
}

var notVisited = _nodes.Except(visited);
_edges.RemoveAll(e => notVisited.Contains(e.Target));
_nodes.RemoveAll(n => notVisited.Contains(n));
```

Výpis 4: Odstranění mrtvých uzlů

```
var visited = new List<Node>(); // seznam navštívených uzlů
foreach (var node in _nodes.Where(n => n.IsFirst)) {
    var stack = new Stack<Node>(); // zásobník
    stack.Push(node);
    while (stack.TryPop(out var n)) {
        if (visited.Contains(n))
            continue;
        visited.Add(n);
        var targets = _edges // cílové uzly všech hran, kde "n" je výchozí uzel
            .Where(e => e.Source == n)
            .Select(e => e.Target);
        foreach(var s in targets)
            stack.Push(s);
    }
}

var notVisited = _nodes.Except(visited);
_edges.RemoveAll(e => notVisited.Contains(e.Target));
_nodes.RemoveAll(n => notVisited.Contains(n));
```

Výpis 5: Odstranění nedostažitelných uzlů



Obrázek 10: NKA, nad abecedou {a, b}, před optimalizací (vlevo) a po ní (vpravo)

7.3 Tvorba regulárních výrazů

Tvorba regulárních výrazů je u většiny pravidel velice triviální. Není tedy třeba v této práci uvádět všechny postupy. Budou zde tedy vysvětleny pouze algoritmy použité v pravidlech `LengthModRule`, `SymbolCountModRule`, `ConjunctionRule`. Implementace takového algoritmu v `NegationRule` také není triviální, ale postup je zcela stejný jako u `ConjunctionRule`.

7.3.1 LengthModRule

Tento postup používá 2 `for` cykly sekvenčně za sebou. První z nich zajišťuje, aby se ve slově objevilo právě tolik libovolných znaků, kolik je zadaný zbytek po celočíselném dělení (nazveme si jej „$vyraz1$“). Druhý vytváří jiný výraz, který udává, že slovo musí obsahovat takové množství znaků, jaký byl zadán základ modula (tomu v této sekci říkejme „$vyraz2$“).

Poté jsou tyto 2 regulární výrazy spojeny následnovně:

$$vyraz1 \cdot (vyraz2)^*$$

7.3.2 SymbolCountModRule

Třída `SymbolCountModRule` používá velice podobný postup, jaký byl popsán v sekci 7.3.1. Tentokrát však není jedno, jaký symbol je zadán. Musí se střídat zadaný symbol s libovolný počtem jiných znaků.

7.3.3 ConjunctionRule

Protože neexistuje jednoduchý postup, jak konjunkcí spojovat regulární výrazy, vytváří se pomocí konečných automatů. Popis implementace tohoto algoritmu je v sekci 7.3.4. Kvůli optimalizaci se vygenerovaný graf ukládá do proměnné, aby se nemusel vytvářet znovu v metodě `GetAutomaton`.

```
public RegEx GetRegEx(Alphabet alpha) {
   if (Rest >= Base) return new RegEx();

   var r = new RegEx();

   for (var i = 0; i < Rest; i++) {
      if (i == 0)
         r = RegEx.CreateAny(alpha.Letters.Where(l => l != Symbol)).Repeat();
      r.And(Symbol).And(
         RegEx.CreateAny(alpha.Letters.Where(l => l != Symbol)).Repeat());
   }

   var baseRegEx = new RegEx();
   for (var i = 0; i < Base; i++) {
      if (i == 0)
         baseRegEx = RegEx.CreateAny(alpha.Letters.Where(l => l != Symbol)).
            Repeat();
      baseRegEx.And(Symbol)
         .And(RegEx.CreateAny(alpha.Letters.Where(l => l != Symbol)).Repeat());

   }
   if (!r.IsInitialized) r.Init(baseRegEx.Repeat());
   else r.And(baseRegEx.Repeat());

   return r;
}
```

Výpis 6: Tvorba regulárního výrazu třídy `SymbolCountModRule`

7.3.4 Převod konečného automatu na regulární výraz

Algoritmus je založen na postupné eliminaci stavů konečného automatu a jeho nahrazování za hranu, přijímací regulární výraz. Musíme k tomu tedy použít ZNKA. Eliminace probíhá podle obrázku 11.

1. Jsou vytvořeny 2 nové stavy. Jeden počáteční, ze kterého vedou epsilon přechody do všech původních počátečních a jeden přijímací, do kterého vedou zase ze všech původních přijímacích.

2. Všechny původní počáteční a přijímací stavy jsou změněny na klasické uzly (nepočáteční a nepřijímací).

3. Automat musí být převeden na ZNKA. K tomu je vytvořen seznam trojic `regExEdges`, obsahující výchozí a cílový stav a regulární výraz, přes který se hranou přechází - ze začátku jimi jsou pouze původní znaky. Tím sice ztratíme možnost mít uzly, které nemají žádnou hranu, ale ty jsou stejně v tomto případě eliminovány při optimalizaci grafu a pokud se

jedná o původní počáteční a/nebo přijímací stavy, pak jsou podle kroku 1 spojeny s novými stavy epsilon přechodem.

4. Začne cyklus pro každý stav z automatu (je označen jako `q`), kromě nových stavů, počátečního a přijímacího.

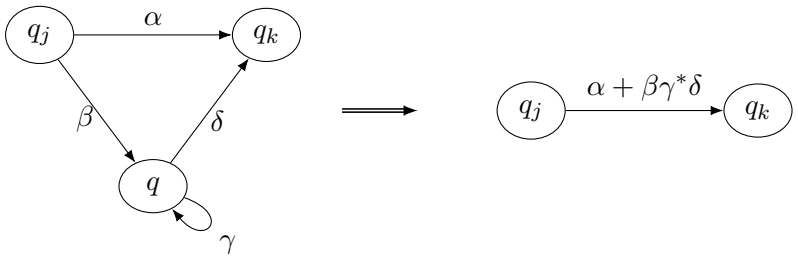
5. Jsou nalezeny všechny stavy, ze kterých vychází hrany do `q` (jsou označeny `qjs`) a stavy, do kterých vchází hrana z `q` (`qks`).

6. Začnou vnořené cykly, které zpracují všechny kombinace každé `qj` z `qjs` a `qk` z `qks`. Uvnitř dochází k samotné eliminaci stavu. Jsou tedy nalezeny veškeré hrany, které odpovídají hraně α, β, γ i δ z obrázku 11. Z těch je poté poskládán regulární výraz $\alpha + \beta\gamma^*\delta$. Před koncem iterace jsou odraněny všechny hrany typu α a je vytvořena nová, se vzniklým regulárním výrazem.

7. Po skončení těchto vnořených cyklů jsou z `regExEdges` odstraněny všechny hrany obsahující `q`, ať už jako výchozí nebo cílový stav.

8. Po ukončení vnějšího cyklu se v `regExEdges` nachází nanejvýš jedna hrana, která je vrácena. Pokud je seznam prázdný, je vrácen prázdný regulární výraz.

[12]



Obrázek 11: Příklad eliminace stavu, při převodu NKA na regulární výraz

7.4 Výpis regulárního výrazu

Regulární výrazy jsou ukládány v upravené stromové struktuře, popsané v sekci 6.2. Aby byl však výraz čitelný, musí být převeden do textové formy. K tomu slouží metoda `ToString` třídy `RegEx`. Využívá k tomu také privátní, rekurzivní metodu `GetSubtreeString`.

Když je uzel typu `VALUE`, pak je vrácena jeho hodnota. Pokud jí je `null`, pak se vrácen znak ε (malé epsilon).

Pokud je však jiného typu, musí být metoda volána rekurzivně, dokud poduzly zpracovávaného uzlu nejsou všechny typu `VALUE`, kdy je vrácena jejich hodnota. Vytvořené řetězce jednotlivých poduzlů jsou spojeny buď operací „$\cdot$“ (pro sjednocení) nebo „+“ (pro zřetězení).

```
if (done.Any(t => t.Item1 == qj && t.Item2 == qk))
   continue;
done.Add(new Tuple<Node, Node>(qj, qk));

if (qj == qk)
   regExEdges.Add(new Tuple<Node, Node, RegEx>(qj, qj, new RegEx(null)));

var alphaRegEx = RegEx.CreateAny(regExEdges.Where(e => e.Item1 == qj
        && e.Item2 == qk).Select(a => a.Item3.Clone() as RegEx));
var betaRegEx = RegEx.CreateAny(regExEdges.Where(e => e.Item1 == qj
        && e.Item2 == q).Select(b => b.Item3.Clone() as RegEx));
var deltaRegEx = RegEx.CreateAny(regExEdges.Where(e => e.Item1 == q
        && e.Item2 == qk).Select(d => d.Item3.Clone() as RegEx));
var gammaRegEx = RegEx.CreateAny(regExEdges.Where(e => e.Item1 == q
        && e.Item2 == q).Select(g => g.Item3.Clone() as RegEx));

var replacement = new RegEx(betaRegEx);
if (gammaRegEx.IsInitialized)
   replacement.And(gammaRegEx.Repeat());
replacement.And(deltaRegEx);
if (alphaRegEx.IsInitialized)
   replacement.Or(alphaRegEx);

regExEdges.RemoveAll(e => e.Item1 == qj && e.Item2 == qk);
regExEdges.Add(new Tuple<Node, Node, RegEx>(qj, qk, replacement));
```

Výpis 7: Část implementace sestavení regulárního výrazu z NKA, popsaná v kroku 6 sekce 7.3.4

Jedná-li se o uzel typu `REPEAT`, pak se za výsledný řetězec přidá „*“. U takového uzlu se počítá s tím, že bude mít pouze jeden poduzel.

7.5 Optimalizace regulárních výrazů

Optimalizace regulárních výrazů probíhá na úrovni samotného stromu ve třídě `Tree`. Slouží k tomu metoda `Optimize`, která je volána ze stejnojmenné metody třídy `RegEx`. Volá své privátní přetížení, přijímací uzel stromu jako jediný parametr. Jedná se o rekurzivní metodu, která je volána pro každý poduzel zadaného uzlu.

Samotné optimalizace jsou však probíhají v metodě `OptimizeNode`, která je volána pro každý uzel stromu. Různé druhy uzlů mají různé optimalizace.

- `AND`:
  - Jsou smazány všechny poduzly typu `VALUE` s hodnotou „`null`“ (reprezentuje epsilon) (např. $\varepsilon a = a$). Pokud poté uzel neobsahuje žádné poduzly, je jeden takový přidán.

- Pokud uzel obsahuje pouze jeden poduzel, pak je jím nahrazen (převezme na sebe jeho typ, hodnotu a poduzly).

- `OR`:
  - Jsou odstraněny všechny poduzly typu `VALUE`, které mají duplicitní hodnoty (např. $a + a = a$).
  - Je-li jeho rodič typu `REPEAT` (má-li nějakého), jsou smazány poduzly typu `VALUE`, které mají hodnotu `null` (např. $(\varepsilon + a)^* = a^*$). Pokud však poté nezůstane v uzlu žádný poduzel, je tam jeden takový zpátky přidán.
  - Pokud uzel obsahuje pouze jeden poduzel, pak je jím nahrazen (převezme na sebe jeho typ, hodnotu a poduzly).

- `REPEAT`:
  - Pokud je jeho poduzel typu `VALUE` a má hodnotu `null`, pak je za něj nahrazen ($\varepsilon^* = \varepsilon$).

8 Nasazení

Program vyžaduje prostředí s nainstalovaným nástrojem .NET core verze 3.1 a vyšší. Byl testován na operačních systémech Windows 10 Pro a Ubuntu 19.10. Testované internetové prohlížeče jsou Opera, Chrome, Firefox a MS Edge. V prohlížeči Internet Explorer se nevykresluje konečný automat.

Aplikace je k datu 11.5.2020 nasazena serveru teoretické informatiky na veřejně dostupné internetové adrese teoretickainformatika.cz/rv, kde je možno si ji vyzkoušet.

9 Závěr

V rámci této práce měla vzniknout webová stránka, která může usnadnit studium formálních jazyků, konečných automatů a regulárních výrazů.

Aplikace obsahuje stručné shrnutí potřebné teorie, podrobné instrukce, jak ji používat a samotný generátor. V něm je možno specifikovat vlastní jazyk nebo si nechat složit náhodný. Administrátor je také schopen přidávat příklady, které si studenti mohou zobrazit například jako referenční řešení.

Toto zadání bylo úspěšně naimplementováno. I přesto by bylo možné aplikaci v pár ohledech rozšířit. Jistě byl vhodný překlad přinejmenším do češtiny. Zde však vziká problém, jak se zachovat k samotnému zadávání pravidel, zda by jejich názvy při specifikaci pravidel měly zůstat anglicky nebo by také měly být přeloženy. Toto by však mohlo být také vyřešeno, vytvořením jednoduššího nástroje pro specifikaci pravidel než zadávání textového řetězce. Například by se mohly pravidla sázet za sebe graficky po kliknutí na příslušné tlačítko.

Práce na tomto projektu byla velice zajímavá, mimojiné protože jsem si vyzkoušel implementaci, ale i samotné vymýšlení různých algoritmů. To mi přišlo velice přínosné, hlavně na procvičení představivosti a *out-of-box* myšlení.

Literatura

1. ZUZAK, Ivan. *FSM2Regex* [online] [cit. 2020-05-08]. Dostupné z: `http://ivanzuzak.info/noam/webapps/fsm2regex/`.
2. JANOŠÍK, Michael. *Konečné automaty*. 2017. Vysoká škola báňská - Technická univerzita Ostrava.
3. HORDĚJČUK, Vojtěch. *Formální jazyk* [online] [cit. 2020-05-13]. Dostupné z: `http://voho.eu/wiki/formalni-jazyk/`.
4. JANČAR, Petr; KOT, Martin; SAWA, Zdeněk. *Deterministický konečný automat: Definice deterministického konečného automatu* [online] [cit. 2020-05-08]. Dostupné z: `http://www.cs.vsb.cz/kot/soubory_animaci/a-definice_dfa.pdf`.
5. JANČAR, Petr. *Teoretická informatika* [online] [cit. 2020-04-12]. Dostupné z: `http://www.cs.vsb.cz/sawa/uti/materialy/ti.pdf`.
6. SAWA, Zdeněk. *Úvod do teoretické informatiky: Regulární výrazy* [online] [cit. 2020-05-08]. Dostupné z: `http://www.cs.vsb.cz/kot/down/uti2007/uti-pr-03.pdf`.
7. MICROSOFT. *Microsoft Docs. Základní dokumentace rozhraní .NET* [online] [cit. 2020-05-14]. Dostupné z: `https://docs.microsoft.com/cs-cz/dotnet/core/`.
8. MICROSOFT. *Microsoft Docs. Dokumentace k ASP.NET* [online] [cit. 2020-05-14]. Dostupné z: `https://docs.microsoft.com/cs-cz/aspnet/core/?view=aspnetcore-3.1`.
9. *Graphviz - Graph Visualization Software* [online] [cit. 2020-05-01]. Dostupné z: `https://www.graphviz.org`.
10. *Michalův web - o informatice, matematice a fyzice: KA03 - Převod NKA na DKA* [online] [cit. 2020-04-18]. Dostupné z: `http://michaluvweb.cz/2015/11/07/ka03-prevod-nka-na-dka/`.
11. SAWA, Zdeněk. *Úvod do teoretické informatiky: Konečné automaty* [online] [cit. 2020-04-12]. Dostupné z: `http://www.cs.vsb.cz/sawa/uti/slides/uti-07-cz.pdf`.
12. KOT, Martin. *Převod konečného automatu na regulární výraz* [online] [cit. 2020-05-13]. Dostupné z: `http://www.cs.vsb.cz/kot/soubory_animaci/a-ka_na_rv.pdf`.

A Přílohy

K této práci jsou dodány také přílohy ve formě ZIP souboru. V jeho kořenovém adresáři se nachází celkem 3 složky:

- **NFA__RegEx__Generator**: Zdrojové kódy samotné aplikace
- **Dokumentace**: Programátorská dokumentace, vytvořená z XML komentářů ve zdrojových kódech ve formátu HTML pomocí programu Doxygen. Lze ji spustit dvojitým kliknutím na soubor „Index.html“
- **Zpracování**: Tento PDF dokument a jeho zdrojový kód v jazyce LaTeX s dalšími potřebnými soubory

Dále se v kořenovém adresáři nachází další ZIP soubor, pojmenovaný „Aplikace.zip“, který obsahuje přeložený a publikovaný projekt.